New Eagle

CONTROL SYSTEM SOLUTIONS

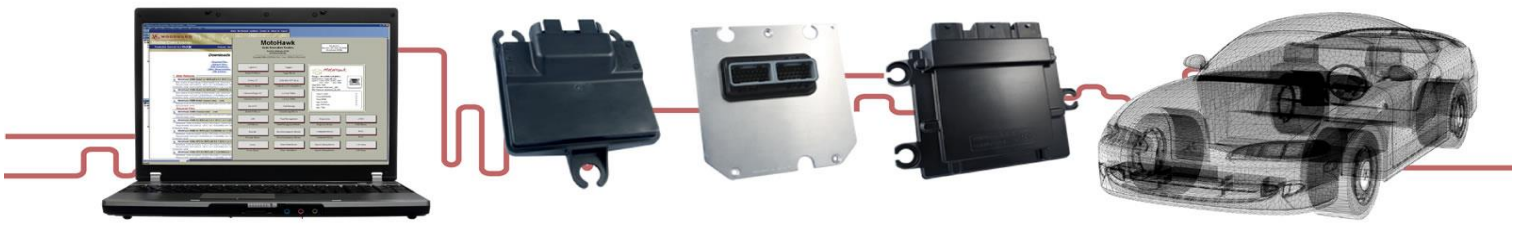
New Eagle Automated Testing

User Manual

Revision 007

(Build 2.0.261.56)

02/17/2014



New Eagle Consulting 3588 Plymouth Road, Box #274 Ann Arbor, MI 48105-2603 PH: 877.234.1410 FAX: 928.395.2114

Contents

1. Introduction

The New Eagle Automated Test (NEAT) system is a PC-based tool that allows developers to write scripted, processor-in-the-loop tests for their Raptor and MotoHawk projects. These tests can be used for regression testing and test-driven development[1].

NEAT can be run either interactively or via the command line, so it is well suited for use in automated build systems such as Cruise Control.

NEAT includes the ability to build models in MATLAB and flash an ECU.

2. Automated Testing System Architecture

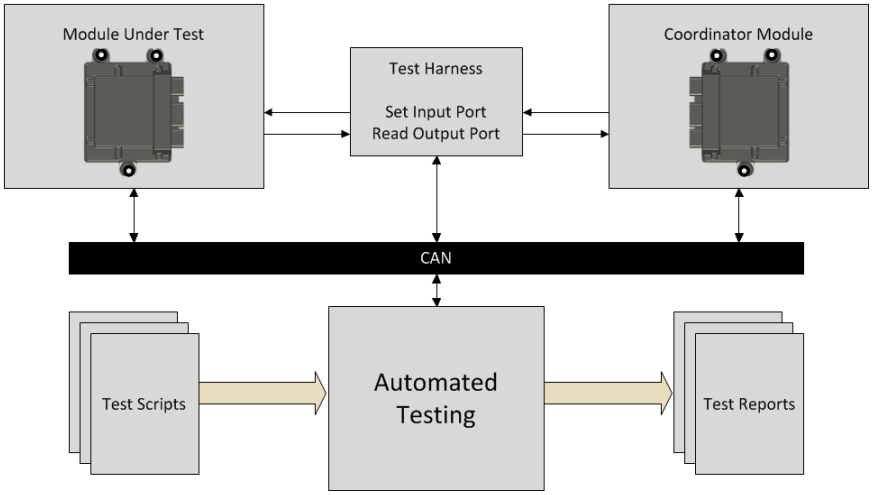


Figure 1 – Automated Testing System Data Flow

The hardware components of a NEAT system are:

1. Windows based desktop computer for running NEAT
2. A cable for connecting the Windows machine to the CAN bus (Kvaser, etc.)
3. The module to test
4. (optional) A "coordinator" module for testing I/O
5. (optional) A wiring harness which connects the coordinator module to the module-under-test

The software components of a NEAT system are:

1. The NEAT test executor software (on Windows based PC)
2. The Raptor or MotoHawk model flashed onto the module-under-test
3. (optional) The coordinator model flashed onto the coordinator module

[1] http://en.wikipedia.org/wiki/Test_driven_development

3. Configuration

A NEAT configuration executable is provided alongside NEAT to allow users to edit settings

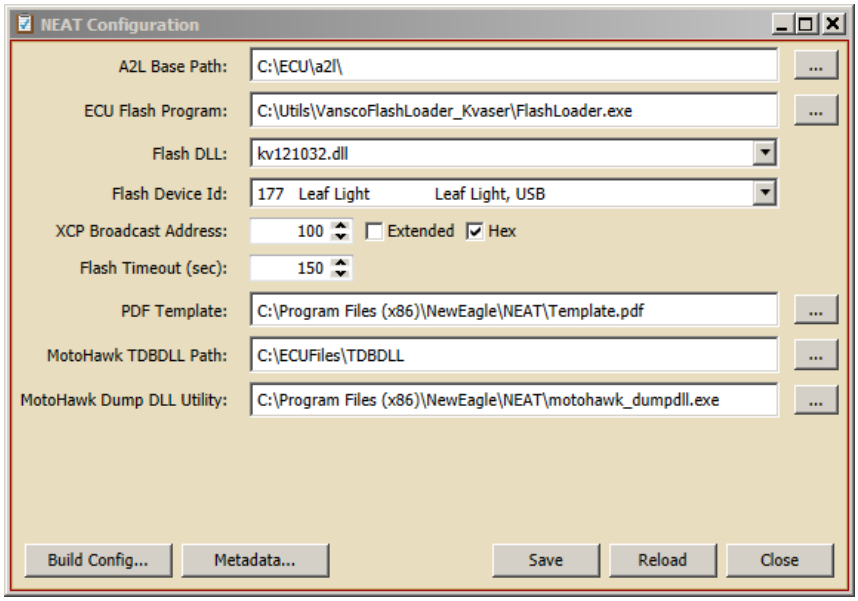


Figure 2 - NEAT Configuration Editor

Note: you must press "Save" before exiting the NEAT Configuration editor or the new configuration settings will not be saved.

3.1 Configuration Settings

- `A2L Base Path`: Directory where .a2l files will be stored when Raptor models are built
- `ECU Flash Program`: Path to the executable used to flash test and coordinator modules
- `Flash DLL`: .dll file used by ECU flash utility
- `Flash Device Id`: ID of device used to flash ECU's
- `XCP Broadcast Address`: XCP Address used to communicate with modules
- `Flash Timeout (sec)`: Number of seconds that NEAT will wait while waiting to flash an ECU (if applicable)
- `PDF Template`: Path to file that NEAT will use as a template when generating pdf reports
- `MotoHawk TDBDLL Path`: Directory where MotoHawk .dll files will be stored when MotoHawk models are built
- `MotoHawk Dump DLL Utility`: Path to MotoHawk Dump DLL Utility

4. Automated Testing Console

The NEAT Console is where you can author, edit, and run tests interactively.

Launch the NEAT Console from the Windows Start Menu or by double-clicking on a NEAT test script in Windows Explorer. If you start NEAT via the Windows Start Menu, you will have the option of manually opening an existing NEAT file, creating a new file, or opening a recently used file.

4.1 Creating an Automated Testing Script

To create a new NEAT script, select the `New` option from the `File` menu. NEAT will show you a dialog box that lets you configure the new test script (See Figure 3 - Create Test File Dialog).

Select the framework for the Test Target module (Raptor or MotoHawk) and the Coordinator module, if any. For both Raptor and MotoHawk, select the SimuLink model file (MDL or SLX) for the software that will be running on the module and select the Matlab version that should be used for automated building. The `Raptor Version` and `MotoHawk Version` fields are optional, and are intended for future use.

For Raptor models:

- Enter the flash address used for flashing the module for targets supporting the Vansco Flash Loader. This is a hexadecimal number ranging from 0x00 to 0xFF (255). Note that as of Raptor 2013b there is no way to change this value in the model from the default of 0xFD (253).
- Enter the XCP address on which the model will receive XCP messages. This can be found and modified in the `XCP Protocol Definition` block in the Raptor model. NEAT uses XCP to communicate with Raptor models, so the model must contain an `XCP Protocol Definition` block to be used with NEAT.

For MotoHawk models:

- Enter the city id for the ECU. This is the value set in the `MotoHawk CAN Definition` block in the MotoHawk model.
- Enter the city id for the tool (NEAT). The default value of this is 0x03, so as to not collide with MotoTune, which uses 0x02.
- Select the access level at which to connect to the module. If the model has custom security settings, an appropriately authorized MotoTune dongle must be present when running scripts with NEAT.

Additional Settings:

- `Verify Time`: This is the default amount of time NEAT will wait before determining that a value does not match the required value in a `Verify` action (see Verify 7.27.2 below).
- `Default Delta`: This is the default value for floating-point equality comparisons.
- `CAN Channels`: This is where the user may add CAN channels to use on the Kvaser device that connects to the test harness.
- `Baud Rate`: This is the baud rate used on the designated CAN channel.
- `Name`: Name of the CAN channel as defined in the model file.

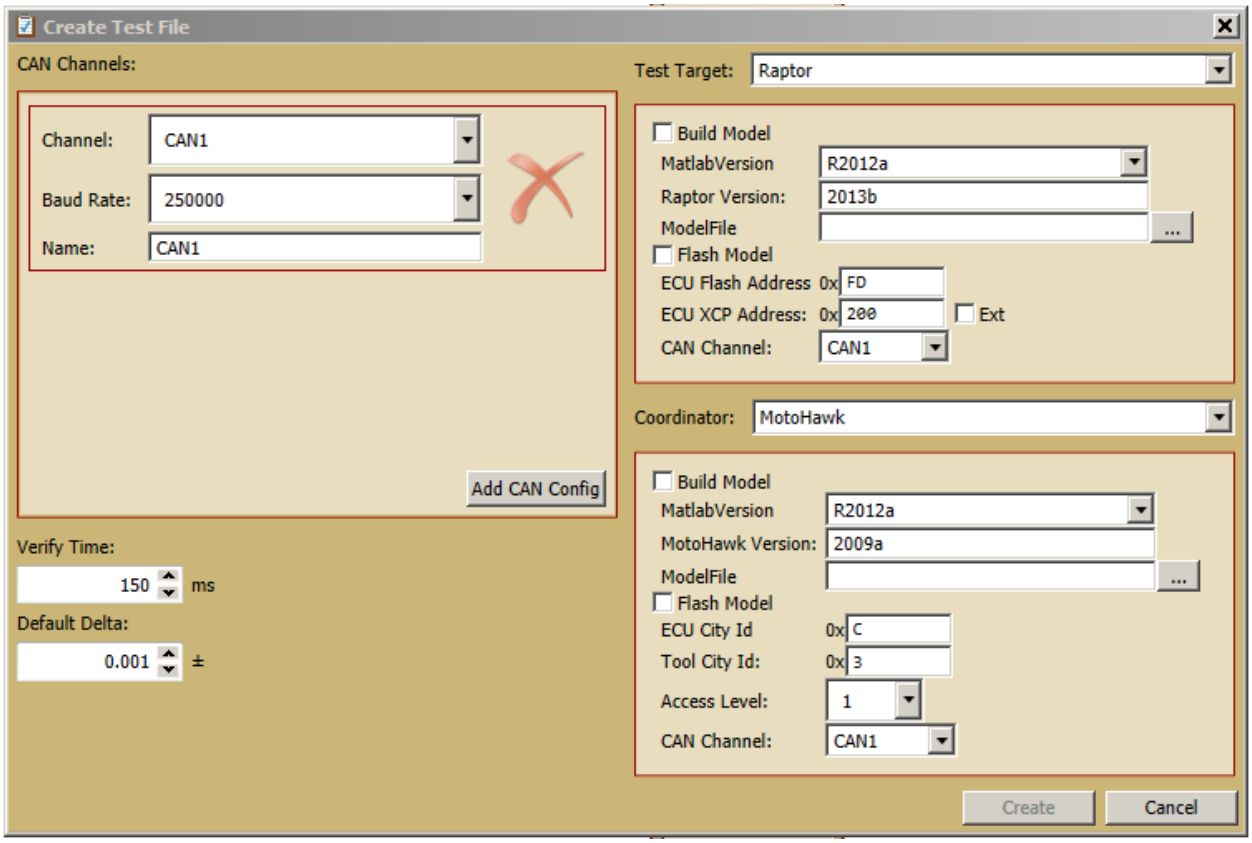


Figure 3 - Create Test File Dialog

Note: These test script settings can be edited later by clicking the button above the navigator pane on the left side of the NEAT console.

4.2 Script Editor

Once the script is loaded, you will be in the test editor, which looks like this:

Figure 4 - NEAT Console Test Editor

There are several important regions on this screen (labeled in blue, above):

- `Test Navigator`: this is a tree that shows all of your tests, groups, and macros. Tests are organized in a nested hierarchy. Search for a test or group by typing part of its name in the `Search` bar.
- `Test Editor`: this is where a test's actions are listed. To add actions to a test, drag the action in from the `Action Toolbox` or a signal in from the `Signals List`.
- `Clean Up Actions Pane`: this area can be expanded by clicking the arrow next to "Clean Up Actions" label. It contains all of the actions that are executed in the clean-up phase of the test. These actions are always executed, even if the test failed or was aborted due to an error.
- `Signals List`: This lists all of the signals available in the ECU. If both a Test Target and Coordinator module are configured, then there will be a drop-down at the top of the Signals List that allows you to switch between the two modules. Search for a signal by typing part of its name in the `Search` bar. To set or verify a signal value, drag the signal name onto the `Test Editor`.
- `Action Toolbox`: This contains test actions that can be put into your test that do not deal directly with an ECU signal. See Section 7: Test Actions, below (page 20), for a description of the operation of each action.

4.3 Test Navigator

The Test Navigator allows you to organize your tests and create new tests, groups and macros. The toolbar at the top of the navigator contains several buttons:

Icon	Name	Function
	New Group	Create a new group under the current group. If the `Tests` node is selected, creates a new root group.
	New Test	Creates a new test under the current group.
	New Macro	Creates a new macro
	New IC Macro	Creates a new Initial Condition Macro
	Run One	Runs the selected test in the Test Runner (see section 4.5: Test Runner, page 14)
	Run All	Runs all of the tests in the Test Runner
	Delete	Deletes the currently selected group, test, or macro. If a group is deleted, all tests and groups under that group are also deleted.

To view a test or macro, simply select it in the tree view and the test's or macro's actions will be displayed in the `Test Editor`.

4.4 Test Editor

This is where you can view and edit what occurs during a test. Actions can be reordered by dragging them up or down. To delete an action, simply select it in the `Test Editor` and press the Delete key.

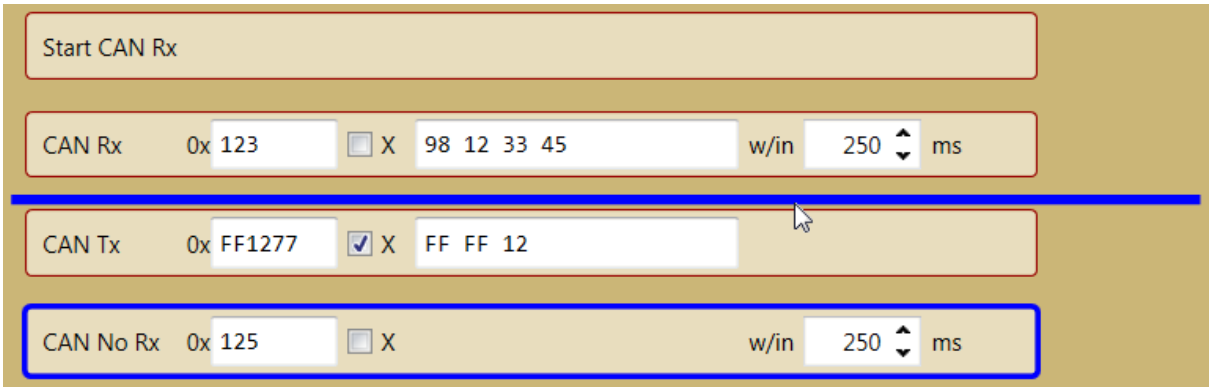


Figure 5 - Moving an Action

4.4.1 Signal Actions

To set a value in the ECU, find the signal in the Signals List. Drop the signal where you want the action to occur in the action sequence. A small pop-up will appear where you can indicate if you want to set the value or verify the value.

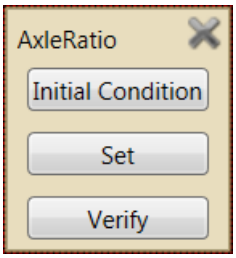


Figure 6 - Signal Actions

`Set` or `Verify` actions can occur anywhere in the test, but `Initial Condition` actions must occur at the start of the test. NEAT will move `Initial Condition` actions to the start of the test to prevent an invalid sequence.

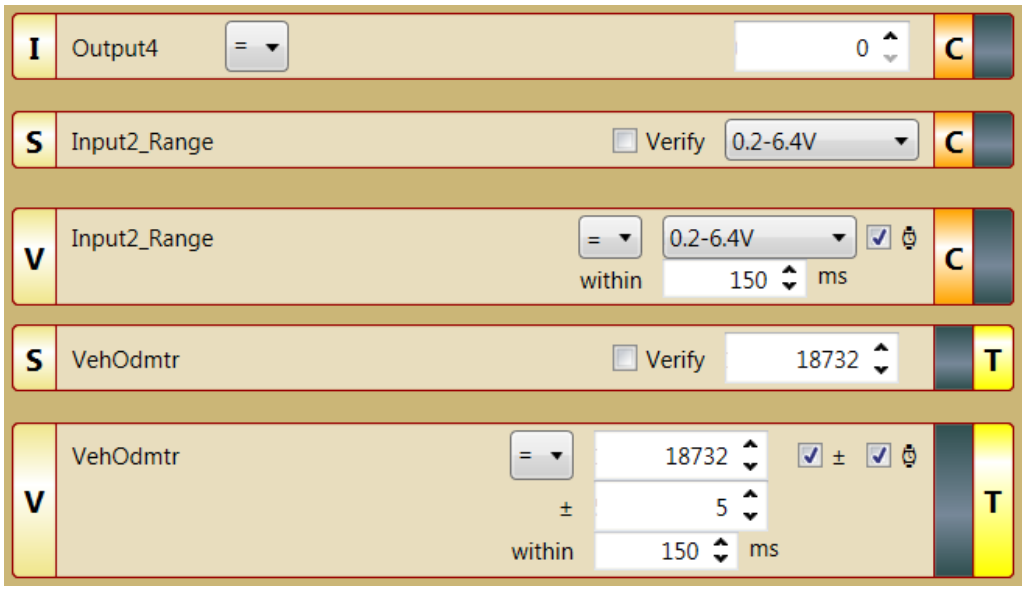


Figure 7 - Signal Actions

The signal action blocks contain a lot of information in a small amount of space:

- The `I`, `S` and `V` labels on the left stand for “Initial Condition,” “Set,” and “Verify,” which is the action type.
- The `C` and `T` labels on the right stand for “Coordinator” and “Test Target”. They indicate which ECU has the signal. If there is no coordinator module, these labels are not displayed, as all signals are on the test target.

- `Initial Condition` actions have an operation selector, where you can choose =, <, >, and ~= (not equal) and an expected value.
- `Set` actions have a `Verify` check box, if checked NEAT will perform a read after the write to verify that the value was written properly.
- `Verify` actions have an operation selector, where you can choose =, <, >, and ~= (not equal) and an expected value.
- `Verify` actions have a ⏲ check box, which optionally shows the `Verify Time` for this action. It is set to the default value specified when the script was created to start, but can be changed as needed for a test. When checked, the verify time is displayed as a text box below the expected value with the label `within`.
- `Verify` actions can have a ± check box (when appropriate), which optionally shows the `Delta` for the operation. It is set to the default value specified when the script was created to start, but can be changed as needed for a test. When checked, the delta can be seen below the expected value.
- The signal referenced in `Set/Verify` actions can be changed by double-clicking the signal name.

4.4.2 CAN Actions

NEAT supports sending and receiving raw CAN frames. These frames are sent or received from the CAN channel set when you created the script.

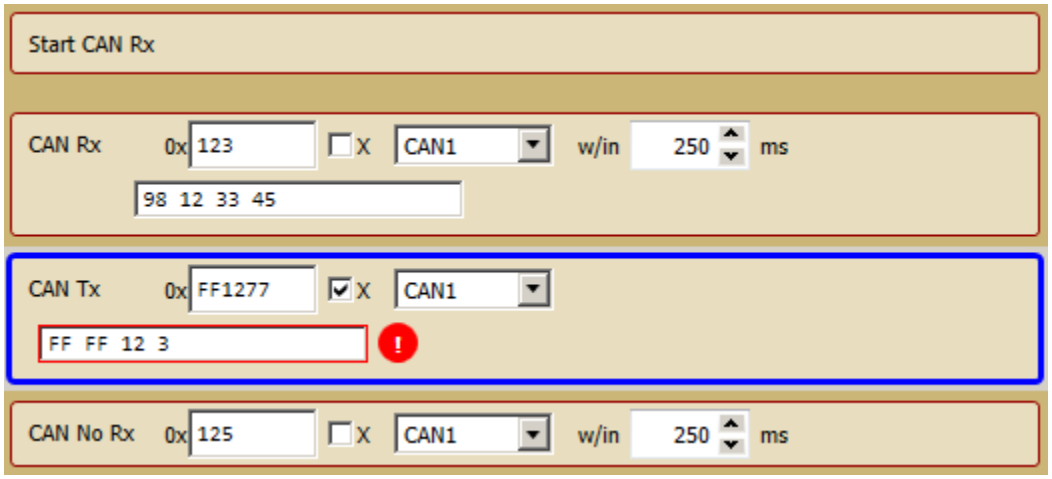


Figure 8 - CAN Actions

CAN frame ids are specified as hexadecimal with no prefix (e.g. - 0x or $). To use an extended CAN id, check the `X` checkbox.

CAN data is specified as a series of bytes in hexadecimal, again with no prefix.

- Each byte is represented by two digits, so zero pad small values (e.g. for the value 11 (decimal) enter “0b”).

- Each byte is separated by a single space.
- If NEAT detects an error in the data format, it will show the error notification displayed above on the `CAN Tx` action (in this case, the 4th byte is missing a digit).
- The message length (either transmitted or expected) is set to the number of bytes specified (so the length for the `CAN Rx` action, above, would be 4).

There are four CAN actions:

- `Start CAN Rx` tells NEAT to start recording CAN frames. NEAT does not record CAN frames unless it has reached a CAN action. Normally this works fine and reduces clutter in the test execution log, especially if the CAN bus is busy. However, in some circumstances, the trigger for transmitting a CAN message in the target model occurs before the execution of a CAN action in NEAT (e.g. - a CAN message is only transmitted when a tracked value exceeds a threshold). In that case, NEAT may miss the message and the test may incorrectly fail. To prevent that scenario, place a `Start CAN Rx` action before the action that triggers the CAN message to start transmitting.
- `CAN Rx` verifies that a particular CAN message is received within the specified time frame. Both the CAN id and the data payload must match in order for this action to pass.
- `CAN Tx` transmits a CAN message on the bus. This message is only sent once (no repeating messages).
- `CAN No Rx` verifies that a CAN message is not received within the specified time frame. Just the CAN id is checked for this action (the data payload is ignored).

4.4.3 Macro Actions

There are two kinds of macros supported by NEAT: `Macros` and `Initial Condition Macros`.

- `Macros` can contain any action other than an `Initial Condition Action` or `Initial Condition Macro Action` (this means that macros can execute other macros).
- `Initial Condition Macros` can only contain `Initial Condition Actions` or `Initial Condition Macro Actions`.

To call a macro, drag `Macro` or `Initial Condition Macro` in from the `Toolbox`. A dialog will be displayed showing defined macros of the appropriate type.

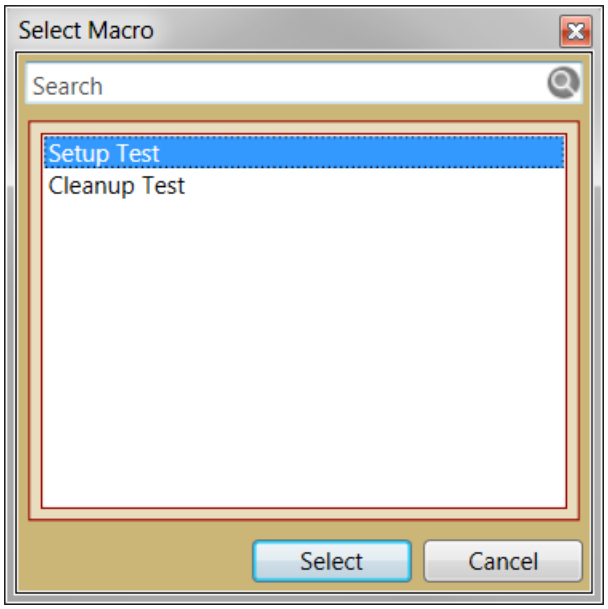


Figure 9 - Select Macro Dialog Box

If you have lots of macros, you can enter part of a macro's name in the search box to narrow down the listed macros.

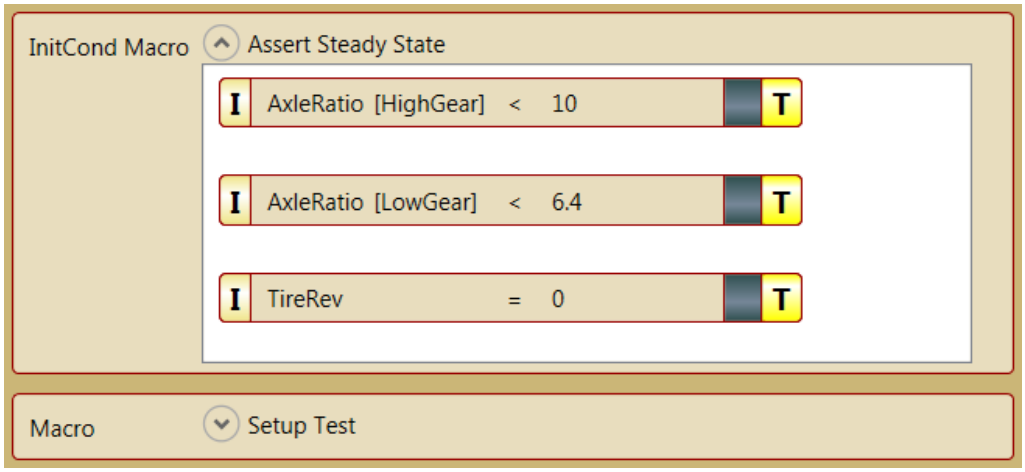


Figure 10 - Macro Actions

Both types of macro actions have an arrow that you can click to see the actions in the macro. You cannot modify the actions in a macro directly in a test. In order to edit a macro, select it from the `Test Navigator`. Any changes made a macro will be reflected wherever it is called.

4.4.4 Other Toolbox Actions

There are several other actions that are useful for composing tests, though they do not interact with the ECU directly.

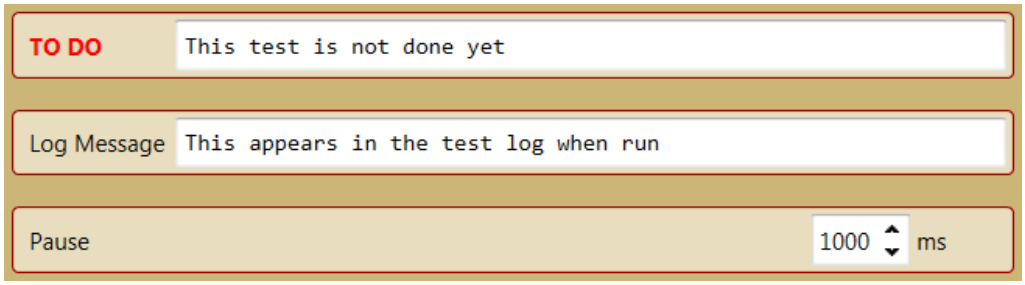


Figure 11 - Utility Actions

The `To Do` action is always placed at the start of the test and will always fail. It is intended to flag tests in progress and prevent one or more of the ECUs being put into an unknown state, which could cause other tests to fail unexpectedly.

The `Log Message` action always passes. It simply inserts a message into the test run log. This is useful for explaining complicated behavior or non-obvious expectations.

The `Pause` action always passes, pausing test execution for the specified number of milliseconds. **Note:** NEAT does not guarantee specific execution time, so the actual delay may be ±50 ms (or more, depending on the host system). If you need to measure precise timing, a better method is to implement that logic in the coordinator module and read the results from NEAT.

4.5 Test Runner

To run multiple tests, switch from the editor view to the `Test Runner` by clicking on either the `Run One` (▸) or `Run All` (⏩) buttons in the `Test Navigator` toolbar.

If the test script contains model building or flashing setting, then the “Build Model” and “Flash Target ECU” sections in the sidebar will contain the relevant information. Unchecking either box will cause NEAT to skip that step when running the tests. Note that if the “Build Model” section is enabled, the “Flash Target ECU” section cannot be disabled.

If a model is built, then the output of the build is scanned to determine which file to use to flash the ECU. The file that is explicitly listed in the flash section is ignored, though the rest of the flash settings are used.

Test groups and individual tests can be toggled on or off for an individual test run by checking or unchecking the box next to the test or group.

To run the selected tests, click the “Run Test” button in the sidebar.

During a test run, the list of tests will be updated as tests are run and pass or fail. The progress bar at the top of the window indicates how many tests remain to run.

If any reference values were included in the scripts, the read values will be displayed in the sidebar once they are read at the start of the test run.

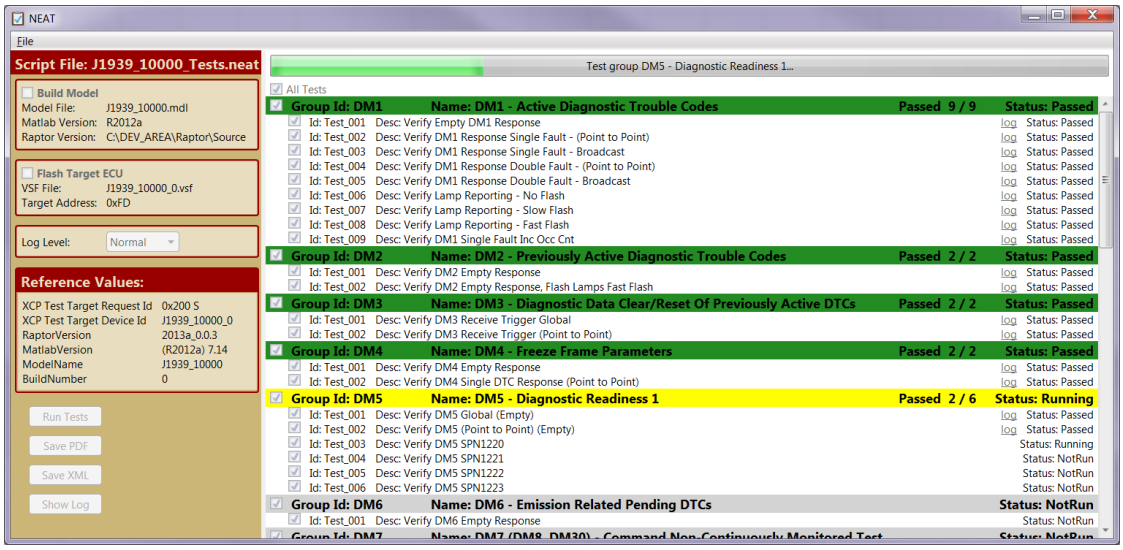


Figure 12 - NEAT Console (during test run)

Once the test is complete, there are several options for examining the results of the run:

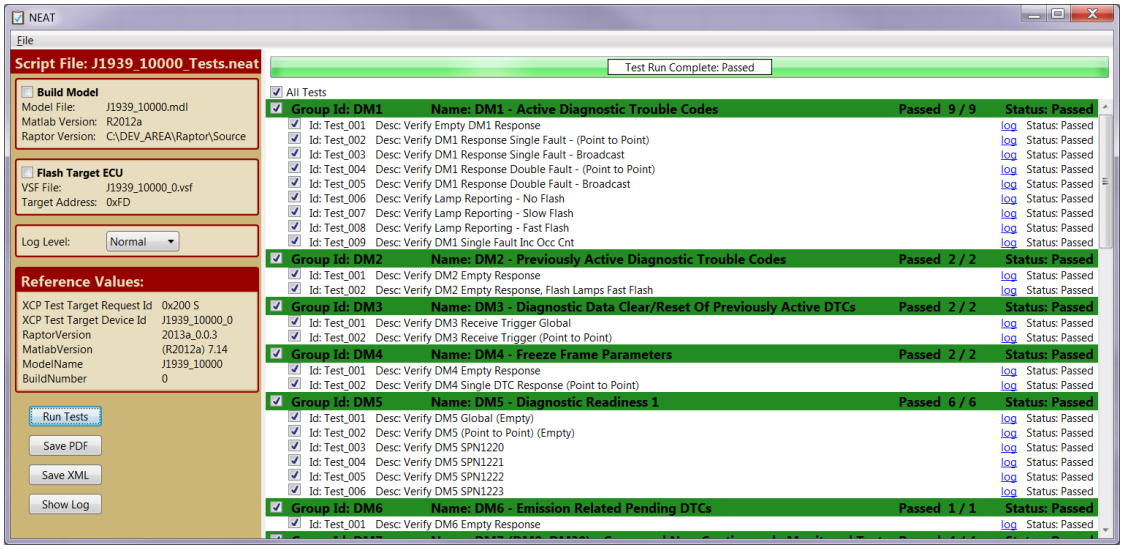


Figure 13 - NEAT Console (after test run)

The full results can be exported in XML or PDF format with the “Save XML” and “Save PDF” buttons, respectively.

The “Show Log” button will display the session log. This log does not contain any information about individual tests, but will be useful for diagnosing things like CAN connection issues and MATLAB build errors.

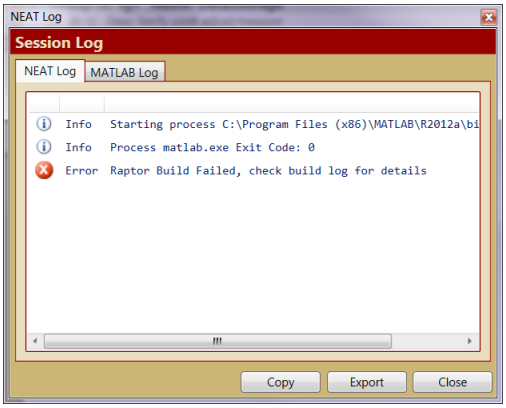


Figure 14 - Log Dialog (NEAT Log)

If there is a MATLAB build error, then consult the “MATLAB Log” to see the build error.

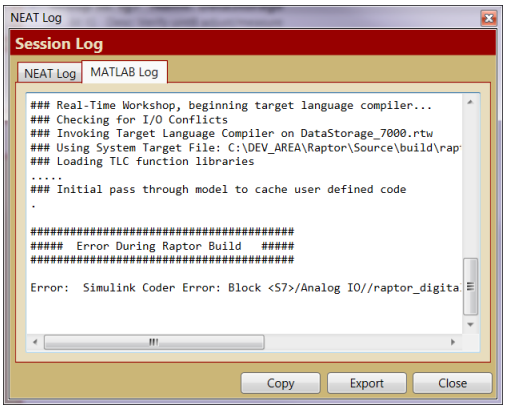


Figure 15 - Log Dialog (MATLAB log)

The log from each test is viewable by clicking on the “log” link next to the test.

5. Command Line Interface

When NEAT is invoked from the command line, there are various command line arguments to control its behavior. This mode is useful for automating test execution. NEAT is found in one of two places, depending on the host OS:

- For 32-bit Windows: `C:\Program Files\NewEagle\NEAT\NEAT.exe`
- For 64-bit Windows: `C:\Program Files (x86)\NewEagle\NEAT\NEAT.exe`

Option	Description	Required
`-t`	Test file	Yes
`-r`	XML report file	No*
`-p`	PDF report file	No*
`-s`	Short PDF report (ignored if `-p` is omitted)	No
`-l[N]`	Logging level (defaults to 0): 0. Normal logging 1. Trace logging 2. Trace and Communications logging	No
* At least one of `-r` or `-p` must be specified		

Figure 16 - Command Line Options

Examples:

Generating an XML report:

```
NEAT.exe -t C:\tests\test.xml -r C:\results\test_results.xml
```

Generating a PDF report:

```
NEAT.exe -t C:\tests\test.xml -p C:\results\test_results.pdf
```

Running with full logging:

```
NEAT.exe -t C:\tests\test.xml -r C:\results\test_results.xml -l2
```

6. Test Scripts

NEAT test scripts are XML files that conform to the `NEATTestScript` or `NEATTestScriptV2` schema. The XSD files for these schemas can be found installed with NEAT.

A test script consists of several sections, in order:

- Test Target configuration (build, flash, and connection settings)
- (optional) Test Coordinator configuration (build, flash, and connection settings)
- Included script files
- Test run reference values
- Macro definitions
- Test definitions

6.1 Test Target and Coordinator Settings

Each of the ECUs can be configured separately, including building the model, flashing the ECU, and connection information (XCP or ERI). The coordinator ECU is optional. For both

ECUs, the build and flash settings are optional. The two ECUs do not have to use the same framework (MotoHawk or Raptor).

6.1.1 Model Building Configuration

The `RaptorModel` and `MotoHawkModel` elements indicate which version of MATLAB and Raptor or MotoHawk to use to build the model file. It also indicates which model file to build.

NOTE: The Raptor/MotoHawk version option is not currently supported. The model will be built with whichever version of Raptor or MotoHawk is configured to run with the indicated version of MATLAB.

This section is optional. If no model should be built for the tests, simply omit the `Model` element entirely.

6.1.2 Module Flash Configuration

The `VanscoFlash` and `MotProg` elements indicate which VSF or SR file to use when flashing the module-under-test. For Raptor projects use `VanscoFlash`; for MotoHawk projects use `MotProg`. Note that when a model is built by NEAT, the MATLAB output is parsed to find the file to flash onto the ECU and any file specified in those elements is ignored.

For `VanscoFlash`, the target source address is set as an attribute. This allows multiple modules to be on the same CAN bus as the module-under-test.

For `MotProg`, only the source file is specified. The connection settings will be taken from the `EriConfig` element.

This section is optional. If no file should be flashed for the tests, simply omit the element entirely. However, if a `MotoHawkModel` or `RaptorModel` element is supplied, then the test script must also contain a corresponding `MotProg` or `VanscoFlash` element.

6.1.3 XCP Slave Configuration

The `XcpConfig` element sets the XCP request id for the module-under-test. All requests to read or write values from the module are sent to this id. It is important that each ECU on the CAN bus have a different request id, but the same broadcast id. For example:

ECU	Request Id	Broadcast Id
Module Under Test	0x100	0x200
Coordinator Module	0x101	0x200

Figure 17 - Sample XCP Configuration

The XCP broadcast id is an application setting, which can be configured with the NEAT Configuration tool.

6.1.4 ERI Configuration

The EriConfig element sets the source (tool) city id, destination (ECU) city id, and access level to use for communicating with the ECU. It is advisable to use a source id other than 0x02, so as to not interfere with MotoTune. If the test contains both a module-under-test and coordinator, the destination city ids should be different, but the source city ids should be the same:

ECU	Source City Id	Destination City Id
Module Under Test	0x03	0x0b
Coordinator Module	0x03	0x0c

Figure 18 - Sample ERI Configuration

6.1.5 CAN Configuration

The `CanConfig` element configures the CAN connection. Set the CAN channel and baud rate for use with this ECU. NEAT currently only supports one configuration for both ECUs, so any CAN configuration set for the Test Target takes precedence over a CAN configuration set for the Coordinator. Future releases may allow different CAN configurations for the two ECUs (e.g. - connect to the Test Target over CAN1 and the Coordinator over CAN2). If no CAN configuration is included, then the default values of CAN1 at 250,000 baud will be used.

6.2 Reference Values

There can be as a many `ReferenceValue` elements as is desired. Each is a value that will be read from the module-under-test at the start of the test run and reported in the XML and PDF reports.

6.3 Included Script Files

Use the `Include` element to import the contents of other script files. Each file is included only once, even if it is referenced multiple times, and can include additional scripts through the use of the `Include` element. Everything in an included script is imported, except the

`Model` and `Flash` elements, which are ignored in all scripts except the original script. Macros, test groups, etc., all must have unique names or ids across all scripts included.

6.4 Macro Definitions

There are two types of macros: normal macros and initial condition macros. Both are named sets of actions that are run in sequence with the other actions in a test. Each type of macro can contain references to other macros of the same type. These actions are executed in order when the macro is referenced in a test. There is no support for parameters or return values.

6.5 Test Definitions

Tests are grouped into a hierarchy under `TestGroup` elements.

A test consists of:

- (optional) Initial condition verification
- A sequence of actions that perform the test
- (optional) clean up actions

During a test run, if any action fails or encounters an error, the test fails and execution is aborted. If there is a `CleanUp` section defined, those actions will be executed even if the test fails.

7. Test Actions

7.1 Set

The `Set` action sets a value in a module. In Raptor, this is done via the XCP protocol and the value must be set up as an Adjustment in the model. In MotoHawk, this is done via the ERI protocol, and the value must be set up as a calibration in the model with an appropriate access level.

Options:

- `Target:` Indicates which module the value should be set in: the module-under-test or the coordinator module. If no target is specified, then the value will be set in the module-under-test.

Examples:

```
<Set Adjustment="CompleteDriveCycle" Value="1"/>
<Set Adjustment="CompleteDriveCycle" Value="1" Target="TestTarget"/>
<Set Adjustment="Target_DigInput1" Value="1" Target="Coordinator"/>
```

7.2 Verify

Reads a value in a module and compares it to an expected value. In Raptor, this is done via the XCP protocol and the value must be set up as an Adjustment or Measurement in the model. In MotoHawk, this is done via the ERI protocol and the value must be set up as a calibration or probe in the model with an appropriate access level.

Options:

- `Target`: Indicates which module the value should be read from: the module-under-test or the coordinator module. If no target is specified, then the value will be read from the module-under-test.
- `VerifyTime`: Indicates how long, in milliseconds, NEAT should wait for the value in the ECU to match the expected value. NEAT will poll the value approximately every 50 ms. If at the end of this time the value in the ECU does not match the expected value, then the test will fail. If this option is not specified, NEAT will use the default value specified on the root `NEATScript` element.
- `Delta`: Indicates how close a floating-point value can be to the expected value and be considered equal. This is only applied to floating point values; integer and string values must be an exact match. If this option is not specified, NEAT will use the default value specified on the root `NEATScript` element.

Examples:

```
<Verify Measurement="OilPressure" Value="12"/>
<Verify Measurement="OilPressure" Value="12" VerifyTime="200"/>
<Verify Measurement="OilPressure" Value="12.4" Delta="0.01"/>
<Verify Measurement="Target_DigOutput" Value="1" Target="Coordinator"/>
```

7.3 CANTx

Sends a raw CAN message onto the bus. Multi-packet is not supported, so if a long message needs to be sent, include multiple `CANTx` elements. Standard and Extended CAN ids are supported. CAN ids are always specified in hexadecimal, even if the “0x” prefix is omitted.

Message data is specified as a series of two-digit hexadecimal values, one for each byte of data. The length of the message is determined from the number of bytes specified. The data will show up in Kvaser CANKing exactly as it is written in the `CANTx` element.

Examples:

```
<CANTx Id="0x131" Ext="0" Data="01 ba"/>
<CANTx Id="0x3000" Ext="1" Data="12 34 56 78 90 ab cd ef"/>
```

7.4 VerifyCANRx

Waits for a CAN message to be transmitted on the bus. A message id and the expected message data are specified. The first message that matches the id is assumed to be the intended message, and that message's data is checked against the expected data. If the data does not match, this action fails. If a message with the expected id is not received within the amount of time specified in `VerifyTime`, then this action fails.

Examples:

```
<VerifyCANRx Id="0x150" Ext="0" Data="12 34 56 78" VerifyTime="200"/>
<VerifyCANRx Id="0x3000" Ext="1" Data="" VerifyTime="200"/>
```

7.5 VerifyCANNoRx

Waits for a CAN message with the specified id to be transmitted on the bus. If the message is transmitted, this action fails. If no such message is received, then execution continues. No data is specified for this action, it simply checks for receipt of any CAN message with the specified id. The amount of time to wait is specified with the `VerifyTime` attribute.

Examples:

```
<VerifyCANNoRx Id="0x170" Ext="0" VerifyTime="250"/>
<VerifyCANNoRx Id="0x3200" Ext="1" VerifyTime="1000"/>
```

7.6 CANStartRx

This action forces NEAT to begin processing CAN messages. Normally, NEAT does not begin processing CAN messages until a CAN action (`CANTx`, `VerifyCANRx`, `VerifyCANNoRx`) is executed. This is done for performance and to keep noise in the log down in situations where there is a lot of CAN traffic on the bus. However, in some circumstances, a test needs to verify receipt of a CAN message that may have been sent before NEAT gets to a CAN action. In this case, use the `CANStartRx` action.

Example:

```
<CANStartRx/>
<Set Adjustment="StatusMessage_Enable" Value="1"/>
<VerifyCANRx Id="0x230" Ext="0" Data="12 34 56 78 90 ab cd ef" />
```

Here, CAN processing would not normally start until the `VerifyCANRx` action is executed, but the model may have sent the message immediately upon having `StatusMessage_Enable` set to 1, so NEAT would miss the message. The `CANStartRx` action forces NEAT to begin listening to the CAN bus, so the message will be properly received.

7.7 InitialCondition

This action can only be used at the start of a test. It reads a value from the ECU and compares it against an expected value using an equality operation specified in the action. If the read value does not pass the test based on the equality operation and the expected value, the action fails.

Possible operations are: `IsEqual`, `IsNotEqual`, `IsGreaterThan`, `IsLessThan`

Examples:

```
<InitialCondition Name="SPN1213" Operation="IsEqual" Value="0"/>
<InitialCondition Name="SPN623" Operation="IsGreaterThan" Value="0"/>
```

7.8 InitialConditionMacro

Invokes an initial condition macro by name. All `InitialCondition` actions in the macro must pass in order for this action to pass. The macro may be defined in this script file or in another file that is included.

Example:

```
<InitialConditionMacro Name="LampsAreClear"/>
```

7.9 Macro

Invokes a normal macro by name. All actions in the macro must pass in order for this action to pass. The macro may be defined in this script file or in another file that is included. Macros do not support parameterization or return values, they are simply an easy way to execute common sets of commands without having to copy and paste.

Example:

```
<Macro Name="ClearFaults"/>
```

7.10 Repeat

Executes a set of actions a specified number of times. This action contains other actions. The entire sequence is executed the specified number of times. If any of the actions fail during any iteration, then the `Repeat` action fails.

Example:

```
<Repeat Iterations="3">
      <Macro Name="CompleteDriveCycle"/>
</Repeat>
```

7.11 LogMessage

Enters a message into the log. This is useful for debugging complicated tests. The `LogMessage` action does not support any variables or parameters; it just writes a fixed text

message into the log. The log message always shows up in the log, regardless of what log detail is specified.

Example:

```
<LogMessage>Faults should be clear by now</LogMessage>
```

7.12 ToDo

Enters a message in the log and sets the test status to `NotRun`. This must be the first action in a test. A test with this action is not run, but is not failed. This is useful for test-driven-development, where tests have been written for functionality that is not yet implemented. The test run will not have a `Pass` status if everything else passes, but it will not fail.

Example:

```
<Test Id="Test_001" Desc="Verify Empty DM1 Response">
      <ToDo Message="Waiting on implementation of lamp status"/>
            …
</Test>
```

7.13 Pause

Pauses test execution for a fixed amount of time. The `Time` attribute specifies the amount of time to wait in milliseconds. This action cannot fail.

Example:

```
<Pause Time="2500"/>
```

8. Test Results

The results of a test run are reported in two different forms, based on the command line options used to invoke NEAT or based on what button is used in the NEAT Console:

- An XML report (using the `-r` option)
- A PDF report (using the `-p` option)

Each of these contains the same data, just in different formats.

8.1 Test Status

A test can have one of four statuses depending on what happened during the execution of the test (see Table 1 - Test Statuses). The statuses of tests and subgroups in a test group accumulate to determine the status of the test group. The status of the group is set to the lowest priority status of any test or subgroup in the group. The results of an entire script are compiled in the same manner from the statuses of all test groups.

Table 1 - Test Statuses

Status	Description	Priority
`Passed`	All checks in the test passed and all actions completed without error	4
`NotRun`	The test was not run. This can happen if the test has a `ToDo` action or if the user is running NEAT interactively and chose not to run certain tests.	3
`Failed`	One of the checks in the test failed. When a test has this status, the results will contain details about what caused the failure.	2
`Error`	An error occurred during test execution that rendered the results invalid. This could be a missing value in the ECU, a communications failure, etc.	1

Example:

Test	Status	Status Priority	Accumulated Status
Test 1	`Passed`	4	`Passed`
Test 2	`Passed`	4	`Passed`
Test 3	`Failed`	2	`Failed`
Test 4	`NotRun`	3	`Failed`
Test Group	**`Failed`**	**2**	**`Failed`**

Figure 2 - Sample Test Group Status

Here, `Test 3`'s status of `Failed` has the lowest priority of any test in the group, so the entire group has that status.

8.2 XML Report

The generated XML report adheres to the `NEATScriptResults` schema. The XSD file defining this schema can be found installed next to the NEAT executable.

Each test group and test in the script has a corresponding test group or test entry in the results file.

Within each test result entry is a copy of the log for what happened during the test. The amount of information in the log depends on what log level was set at the command line with the `-l` option or what log level was set in the sidebar in the NEAT Console.

8.3 PDF Report

The PDF contains the status of all tests run. It is built on a PDF template that is installed with NEAT. An alternative PDF template can be set with the NEAT Configuration tool. The PDF template should be a single-page PDF (US Letter size) with header and footers approximately the same size as the default template.

9. Test Writing

9.1 Anatomy of a Test

As a general rule, a test should have four distinct phases:

1. Setup
2. Trigger
3. Verify
4. Clean-Up

The **Setup** phase is exactly that: do whatever needs to be done to get the ECU in the state needed to test the target functionality. This can include setting overrides on ECU inputs, disabling certain periodic messages, setting calibration values, etc. It is good practice to use a macro for setup actions that will be common across multiple tests.

The **Trigger** phase is where the target functionality is triggered. This can be done by setting values in the ECU or by sending CAN messages. In some tests, such as when testing steady-state values, there will not be an explicit action needed to trigger the target functionality.

The **Verify** phase is where the target functionality is tested. This is usually done by reading one or more values from the ECU or by listening for a specific CAN Message.

The **Clean-Up** phase is where the ECU is returned to a known state. This phase is the only one explicitly supported by NEAT tests, because this phase needs to occur regardless of what happens in the other phases. It is good practice to use a macro for clean-up actions that will be common across multiple tests.

Example:

This test is from a set of tests for Raptor's J1939 block library. The model should respond to receiving the 0x18EAFDF9 CAN message with the 0x18FECAFD CAN message.

```
<Test Id="Test_001" Desc="Verify Empty DM1 Response">
      <!-- Setup Phase -->
      <InitialConditionMacro Name="LampsAreClear"/>
      <Macro Name="DisablePeriodicMessages"/>
      <Macro Name="ClearFaults"/>

      <!-- Trigger Phase -->
```

```
        <CANTx Id="0x18EAFDF9" Ext="true" Data="CA FE 00"/>

        <!-- Verify Phase -->
        <VerifyCANRx Id="0x18FECAFD" Ext="true"
        Data="00 FF 00 00 00 00 FF FF" VerifyTime="1250"/>

        <!-- Clean-Up Phase -->
        <CleanUp>
              <Macro Name="ClearFaults"/>
              <Macro Name="EnablePeriodicMessages"/>
        </CleanUp>
</Test>
```

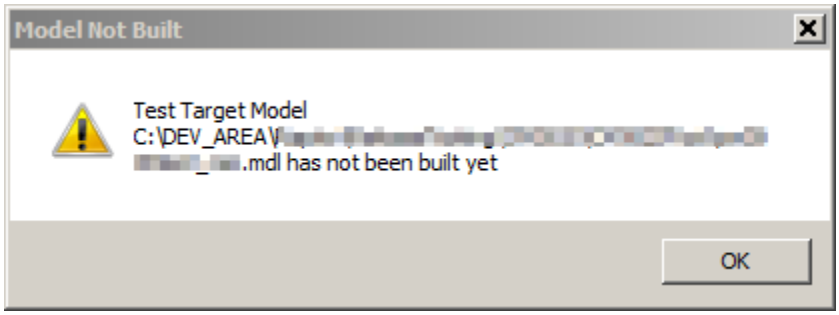
9.2 Test Writing Best Practices

These are guidelines for making tests robust, reliable, and useful.

- Each test should be as narrowly written as possible. Test just one piece of functionality. It is common to have several tests that share common setup and clean-up code and have just a couple of differences in the test actions.
- Each test should be independent. Do not rely on the behavior of other tests, as NEAT does not guarantee execution order or that other tests will be executed at all.
- Use `InitialCondition` actions to verify that the module is in an expected state before running the test. This will help prevent false results as a side-effect of other tests.
- Every value that is modified during a test should be reset to a neutral value during the Clean-Up phase. This will help prevent side-effects that could influence other tests.
- When dealing with CAN messages, it is useful to add hooks to the model that allow periodic messages to be selectively disabled. This will cut down on CAN traffic and makes debugging issues in unrelated functionality easier.
- Tests should be grouped logically. For instance, if the script is testing J1939 functionality, a logical grouping would be one test group for each message type (DM1, DM2, etc.)
- Each test should have a unique, concise description that indicates what is being tested. This will help speed debugging when a test fails.

10. Troubleshooting

10.1 Test/Coordinator Target Model has not been built yet



This dialog will be shown when creating a new NEAT test suite for a Raptor model if the model has not yet been built with Simulink. If the model has been built and this dialog is still showing, make sure verify the NEAT configuration (section 3)